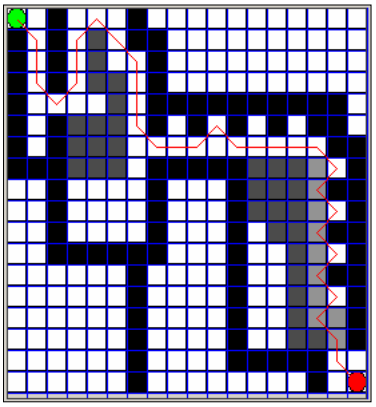
Chapter 3

Decision Making I: Flocking



Overview

In the previous chapters, we looked at how to implement some of the more common pathfinding methods used in today's games. We also touched on the different types of artificial intelligence, which we mentioned included pathfinding, decision making, classification, and life systems. The next type of artificial intelligence we will discuss in the course is *decision making*. In transitioning from pathfinding systems to decision making systems, we have decided to begin with a method known as *flocking*. Flocking is actually an interesting mix of the two concepts whereby decisions are made about how to perform pathfinding based on what the rest of the flock is doing. This discussion will set the stage for other forms of decision making that we will look at in the next chapter.

In this chapter we answer the following questions:

- What is flocking?
- What are the common components of flocking?
- How are the common components of flocking implemented?

3.1 Introduction to Flocking

Flocking is a very popular and commonly used AI grouping concept that has existed in computing for a long time. Flocking systems are often used to simulate the behaviors of schools of fish or flocks of birds. While the individual entities participating in a flock are also known as boids in some algorithms, they will be referred to as entities in this text. We can also refer to a flock as a group. This is done in order to make it easier to extend the terminology to group/squad based behavior later in the course. Group/squad behavior is based on a similar concept, where groups of entities cooperate with one another to make decisions and navigate the environment.

3.1.1 Behavior Based Movement

Flocking is classified as "behavior based movement." That is, there is a set of behaviors which each entity in the group applies to determine its movement. Each behavior may or may not be influenced by entities that are nearby, and each will determine the movement for an entity which is appropriate for the particular behavior. For instance, a "move up" behavior would simply apply a movement vector that takes the entity straight up.

There are four key movement behaviors typically applied in flocking systems. Using these four behaviors will result in fairly realistic representations of flocks of birds or schools of fish. They are also helpful when simulating movement for other entity groupings, as we will see later. We will examine each of these behaviors in detail shortly. First, let us set forth a few standards which we will be using in our examples.

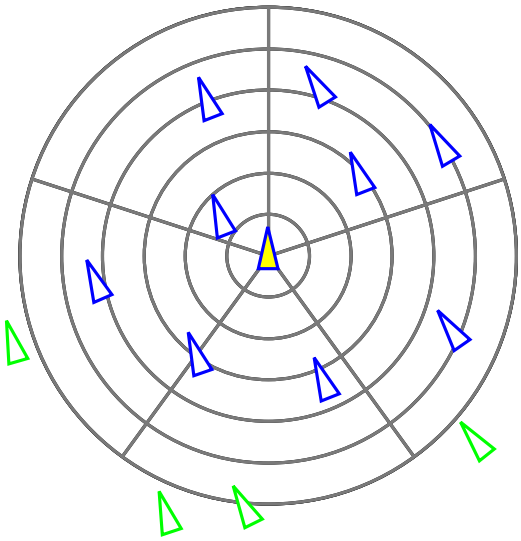


Figure 3.1

Figure 3.1 shows a group of entities represented as colored triangles. The yellow filled triangle in the center represents the entity of interest. Its movement choices will be examined and it will be referred to as "our entity" in this section. The empty blue triangles are other entities which our entity can perceive. The green entities are outside our entity's perception range, and will be ignored by our entity, even though they belong to its group.

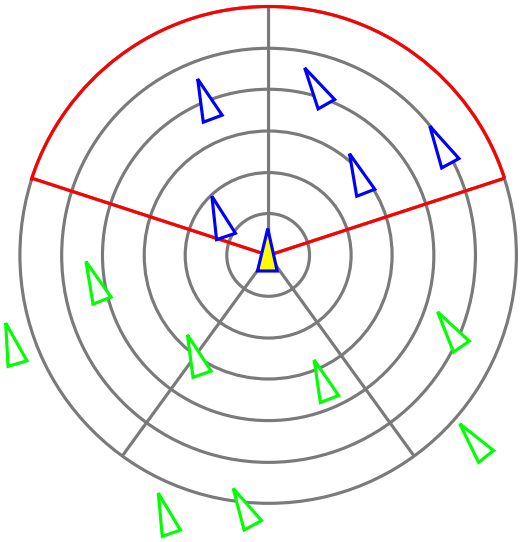


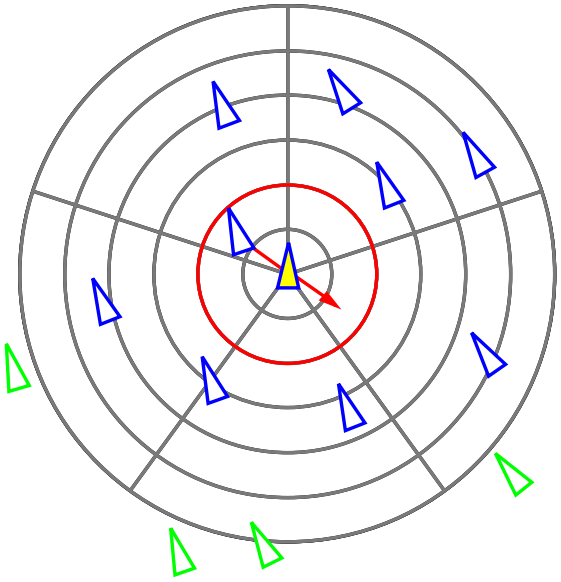
Figure 3.2

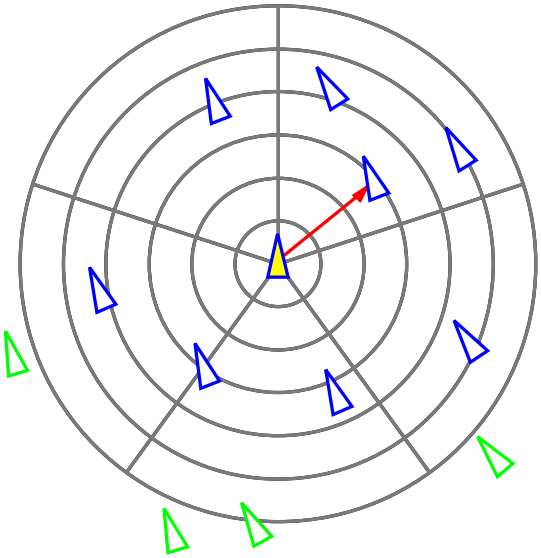
The radial lines on the underlying polar grid will show the relative positions of all other entities with respect to our entity. Note that if desired we can limit the perception of our entity to a cone rather than a full 360 degree area. For instance, if we wanted to reduce our entity's field of view to 144 degrees (the top two arcs), then the four blue entities outside the cone range would become green (Figure 3.2). In Figure 3.2, the red section of the polar grid indicates the area in which our entity can perceive the other members of its group.

Note the concentric rings in the polar grid. These can be used to easily visualize the distance from our entity to the other entities. The separation behavior has a particular interest in this information since it is influenced by distance. With these standards set, let us now take a more detailed look at the behaviors.

3.1.2 Separation

The separation behavior strives to keep the entities in a group from clumping too closely together without letting them drift too far apart. In the figure on the right, we have a group of entities. The blue entities are within our bounds of knowledge, so we pay attention to them, while the green entities are too far away to concern us, so they are ignored. Let us say that in our separation example we want a distance of two grid spaces between each entity. It follows that any entity within the bounds of the red circle is too close. The entity above and to the left (connected by the red line) is too close according to our specifications, so we calculate a vector that takes us away from this entity, as shown by the red arrow.

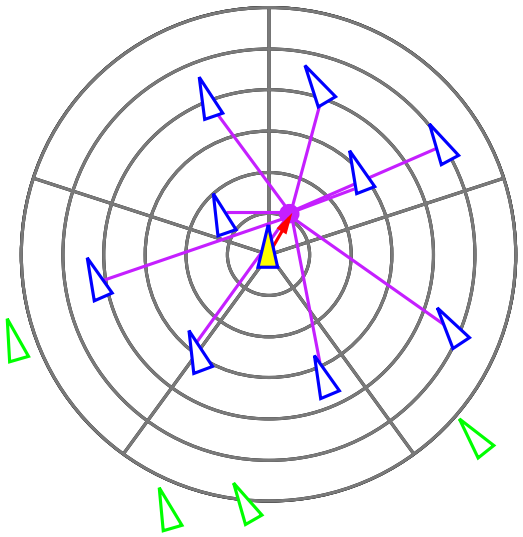




In the figure on the left, there are no entities which fall within the distance specification to our entity. In fact, all of the entities are further away than the second circle. In this case, we choose the closest entity to our entity, and move towards it (as indicated by the red arrow). Thus, the separation behavior strives to keep the separation between entities a certain fixed distance.

In both of these cases, the desired move vector is summed together with the rest of the desired move vectors from the other behaviors. Let us continue on and look at those next.

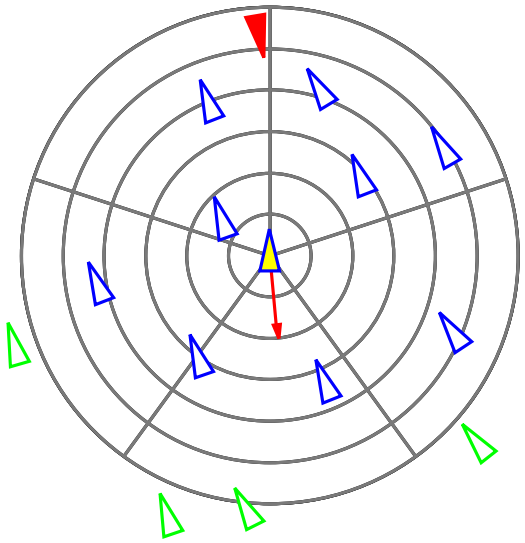
3.1.3 Cohesion



The cohesion behavior tries to emulate the behavior in flocks of birds and schools of fish where they tend to "stick together". The idea is that while we do not want the entities to run into each other, we want them to be in close proximity because this provides a degree of safety. In real schools of fish for example, they stay together because it presents a larger combined front to predators, and conveys the impression that the group as a whole is a larger entity than the predator. In the ocean, size mostly determines who will be eaten by whom. If that concept fails, there is a secondary safety measure which is based on lowering the probability of being eaten if the entity is not alone. For example, if a shark is chasing after two fish, the fish that will likely survive need only swim faster than his brethren.

Take another look at our group of entities. Again, we are concerned with the yellow filled entity in the center of the polar grid; the blue entities are those in the group that are in our range of interest, and the green entities are those outside that range. The cohesion algorithm collects all of the entities within our interest range and computes the collection of entities' average position. We then have our entity move towards this position. As in the case of the separation behavior, this desired move vector is summed up with the other behaviors' desired moves.

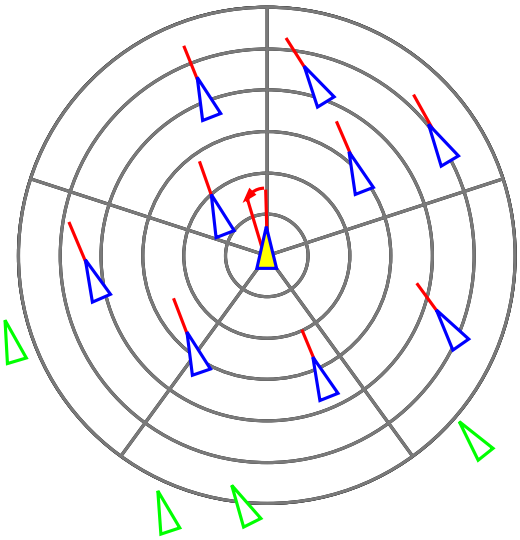
3.1.4 Avoidance



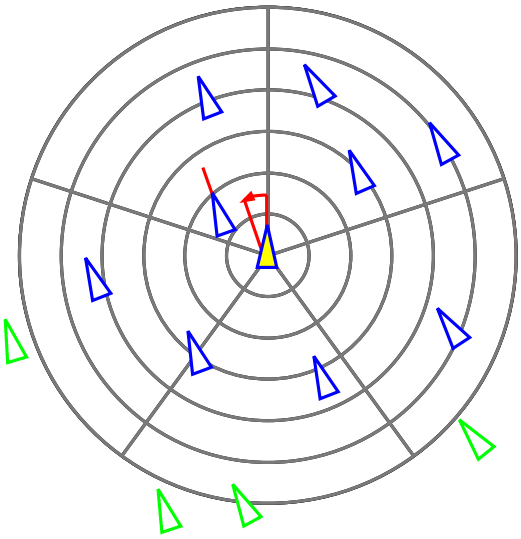
The avoidance behavior is the behavior that directs the group to move away from things they do not want to be in proximity to or come into contact with. Examples for schools of fish include predator fish that will eat them, boats and their whirling propeller blades, and other obstacles which they cannot swim through. The idea is that the entities would sense things other than their group mates, and if the thing they sense is not something they want to get too close to, they move away from it. In the figure on the right, the large red triangle might represent a predator fish. Our entity finds that it is too close to this fish (in this case, sensing a predator fish at all would be considered too close), so the behavior determines a desired movement vector in the direction of the red arrow. This vector will be added in with the rest of the desired movement vectors from all the other behaviors. This behavior is quite interesting because it differs from the others in that it includes concepts like observation and environmental awareness. That is, the entity is paying attention to objects and/or events beyond just itself and its group.

3.1.5 Alignment

The alignment behavior attempts to keep all of the entities in the group aligned in approximately the same direction. As in the real world, all of the fish in a school tend to swim in the same direction.



Consider the figure on the right. Most of the entities are headed in the general same direction, but not exactly. The alignment behavior gathers up the entities in our range, averages their heading, and sets our entity's desired heading to be more like the average heading. This should not be done instantly, since it is not realistic that such a change occurs immediately. Over time, this method will gradually adjust our entity's direction so that it will be in line with the alignment of the rest of the group.



Another way the alignment behavior could work would be to take the nearest entity's heading rather than an average heading of all the nearby group mates (see figure on left). This is computationally less expensive, since we do not have to search through all of our nearby group mates and average up their headings. But it can also result in parts of the group veering off in much different directions since we only pay attention to the entity closest to us.

3.1.6 Other Possible Behaviors

While the four behaviors discussed previously are the mainstay of flocking implementations, there are other behaviors that can be helpful. For instance, a cruising behavior is a useful behavior. A cruising behavior decides what direction the entity would go if that entity were alone. In most cases these behaviors simply maintain the last heading the entity had and add some random variation. There are times where this behavior actually would get called upon even when there are group mates, because sometimes (mostly due to the avoidance behavior making them run away from something) an entity may be separated from its group and forced to fend for itself.

Another example of an extra behavior is the “keep in sphere” behavior used in our demo. This behavior will be covered in more detail shortly, but the basic premise is that, as long as the entity is within the bounds of the sphere, the behavior has no input. However, as soon as the entity leaves the sphere, the behavior puts in a request for a movement towards the center of the sphere.

3.2 The Flocking Demo

In keeping with the spirit of traditional flocking, the demo provided for this chapter consists of several schools of fish. The player takes on the role of a northern pike, hungry and ready to eat. There are schools of blue gill and large mouth bass swimming about, and they will react to the player if he gets too close. If the fish are touched, they will not disappear as if they have been “eaten” (that is left to you to add!). The main goal is to provide a means by which a flocking implementation can be seen, and provide an interface through which the various parameters of each of the behaviors can be adjusted to see the results.

3.2.1 Design Strategies

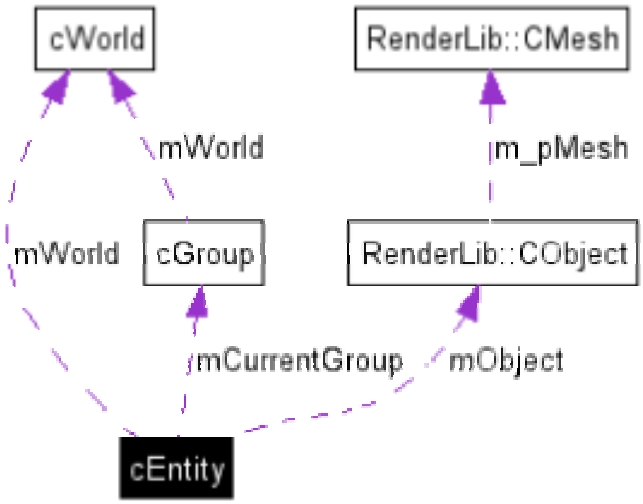


Figure 3.9

The first thing to understand is how everything is laid out in the Flocking Demo. Figure 3.9 shows the collaboration diagram for the cEntity class. The cEntity class is basically a single thing in a group. Other implementations might call it the ‘boid’ in a flock. Every cEntity is in a cGroup, even if it is the only one in the group. All of the cGroups are managed by the cWorld object, of which there is only one. Each cEntity also maintains a connection to its rendered representation as well, but we do not need to go into details about that.

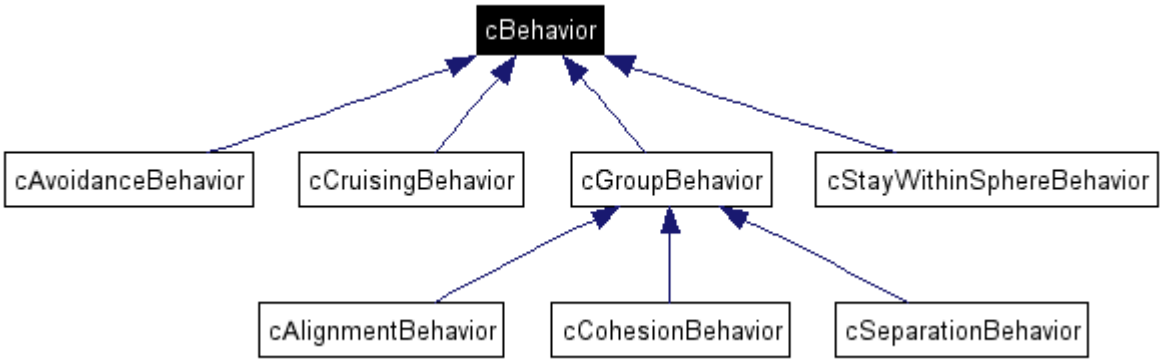


Figure 3.10

Every cEntity has a list of cBehavior objects which it uses to determine how it wants to move around. It does not own these behaviors but shares them with all of the other entities. The behavior itself gets a reference to the current entity when it is applied, to make the behavior shareable. The algorithm does not need to maintain state since the entity can do that. As you will notice in Figure 3.10, not all of the behaviors are group related. Only the Alignment behavior, the Cohesion behavior, and the Separation behavior make use of the group information. The rest of the behaviors solely depend upon the entity being acted upon. We will go over each of these classes in detail shortly.

3.2.2 MFC and our Demo

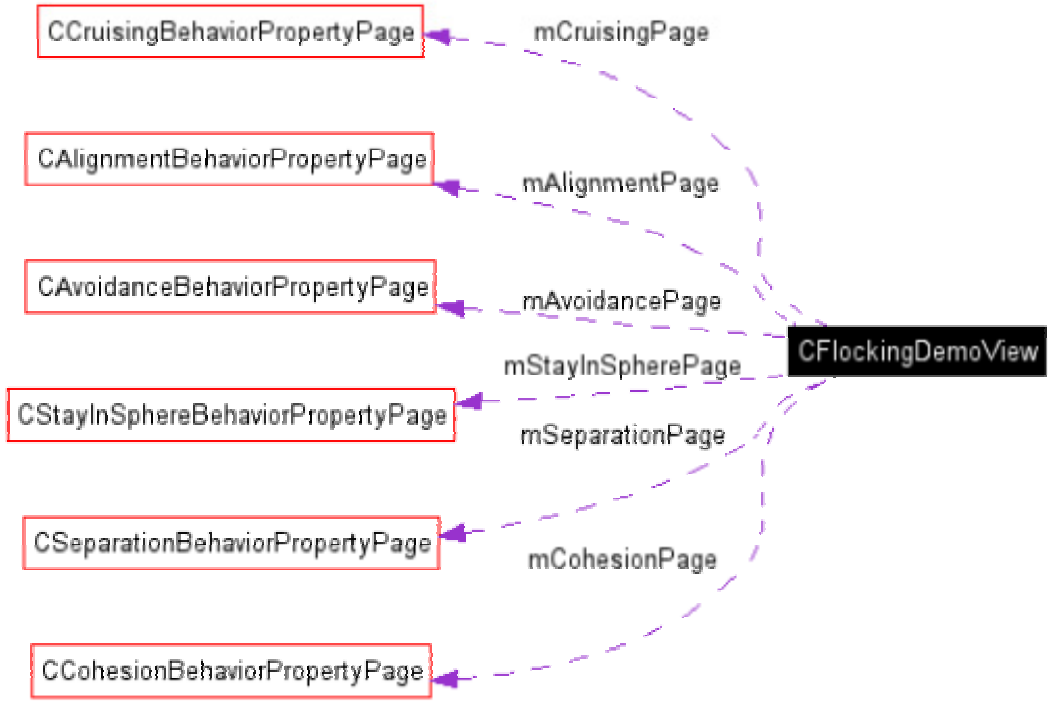


Figure 3.11

As in the Pathfinding Demo, the Flocking Demo makes use of MFC. There is a CFlockingDemoApp which has a current document (CFlockingDemoDoc) and some associated views (CFlockingDemoView and C3DView). In this particular application, the document does not hold anything; the C3DView holds the rendering engine which keeps track of the cWorld object and its associated entities. The CFlockingDemoView, however, does have some things of interest. This view contains a tab control which has all the property panels for each of the behaviors. If you desired to create your own behaviors, this is where you will want to link them up to the interface to have your own property panel. As you can see in Figure 3.11, the CFlockingDemoView keeps track of a collection of pages, one for each of the behaviors. Later we will see that the cEntity objects do not own the behaviors. They all share the same set because the behavior itself does not change. This allows the property panels to modify the behaviors globally for all cEntity objects. Now let us examine the actual implementation of our demo.

3.2.3 Our Implementation

```
class cWorld
{
public:
                                cWorld(void);
      virtual                  ~cWorld(void);

      void                     Add(cGroup &group);
      void                     Remove(cGroup &group);

      tGroupList               &Groups() { return(mGroups); }

      virtual void             Iterate(float timeDelta);
      virtual void             Render(LPDIRECT3DDEVICE9 pDevice);

protected:
      tGroupList               mGroups;
};
```

Above we have the declaration for the cWorld object. There is only one of these in existence at any given time. It holds all of the groups and is responsible for iterating them during each time step. Additionally, the world takes on the responsibility for ensuring that the groups render themselves. Let us go over this in a little more detail.

```
                                cWorld(void);
      virtual                  ~cWorld(void);
```

A default constructor is provided which initializes the group list properly. The group list at construction time is empty. The destructor is virtual to allow for more specific derived classes to be polymorphically destructed properly. The destructor ensures that all of the groups are properly freed.

```
      void                     Add(cGroup &group);
      void                     Remove(cGroup &group);
```

The cWorld object provides the ability to add and remove groups from its list. When a new group is added to the cWorld object, it takes over responsibility for its lifetime, so the group added need not be destroyed externally. If the group is removed, however, responsibility is relinquished and the group must be freed manually.

```
	tGroupList		&Groups() { return(mGroups); }
```

The cWorld also provides an accessor to the list of groups it contains. This allows external objects to iterate across all of the items that exist in the world if necessary. The avoidance behavior makes use of this method.

```
	virtual void		Iterate(float timeDelta);
```

The iterate method advances time by the time delta for each of the groups. Since this is done in an iterative fashion, the items that are last in the list gain the most benefit as they have watched others go before themselves and know more about the true state of the world at the end of a time slice. However, this comes with a potential penalty as they could be eaten before they get their turn.

```
	virtual void		Render(LPDIRECT3DDEVICE9 pDevice);
```

The cWorld is responsible for ensuring that each of the groups is rendered when the time comes. It passes responsibility to each of the individual groups to make sure their contents are rendered correctly.

```
	tGroupList		mGroups;
```

The world owns the list of groups that it holds. It will delete any of the groups in this list, so once a group is added to the world, you do not need to externally destroy it as the world assumes that responsibility.

```
class cGroup
{
public:
				 cGroup(cWorld &world);
	virtual			~cGroup(void);

	void			Add(cEntity &entity);
	void			Remove(cEntity &entity);

	tEntityList		&Entities() { return(mEntities); }

	virtual void		Iterate(float timeDelta);
	virtual void		Render(LPDIRECT3DDEVICE9 pDevice);

protected:
	tEntityList		mEntities;
	cWorld			&mWorld;
};

typedef vector<cGroup*> tGroupList;
```

Here we have the definition of the cGroup class. This class holds one collection of cEntities that will be acting together. It also holds a reference to the cWorld which owns it so that it can gain access to the list of all of the other cGroup objects.

```
                              cGroup(cWorld &world);
      virtual                 ~cGroup(void);
```

The cGroup object provides no default constructor, as it needs a reference to the cWorld object that owns it to build its held reference. The destructor is virtual to allow for correct polymorphic destruction in the event that a new type of cGroup is derived. The cGroup object is responsible for its list of cEntity objects and will destroy them when the destructor is called.

```
      void                    Add(cEntity &entity);
      void                    Remove(cEntity &entity);
```

Similar to the cWorld, the cGroup takes ownership of the cEntity objects added to its list. If the cEntity is removed from the list, the responsibility of releasing the cEntity is relinquished and must be done externally.

```
      tEntityList             &Entities() { return(mEntities); }
```

The group provides access to its list of entities so that the other entities may know of their group mates.

```
      virtual void            Iterate(float timeDelta);
```

The iterate method passes time for all of the entities in the group's list. This method will be called by the cWorld object that holds the cGroup.

```
      virtual void            Render(LPDIRECT3DDEVICE9 pDevice);
```

The cGroup is responsible for ensuring that all of the entities owned by it are rendered when the time comes. This method will be called by the cWorld that owns the cGroup.

```
      tEntityList             mEntities;
      cWorld                  &mWorld;
```

The cEntity objects contained in the mEntities vector are all owned by the cGroup object, and will be destroyed upon destruction of the cGroup object. The cWorld reference is made available so the cGroup can gain access to all of the other cGroup objects contained in the cWorld object.

```
class cEntity
{
public:

      cEntity
      (
            cWorld &world,
            unsigned type,
```

```
            float senseRange,
            float maxVelocityChange,
            float maxSpeed,
            float desiredSpeed,
            float moveXScalar,
            float moveYScalar,
            float moveZScalar
      );

      virtual                       ~cEntity(void);

      virtual void                  Iterate(float timeDelta);
      virtual void                  Render(LPDIRECT3DDEVICE9 pDevice);

      void                          SetFriendMask(unsigned mask);
      unsigned                      FriendMask(void);
      unsigned                      EnemyMask(void);
      unsigned                      EntityType(void);
      tEntityDistList               &VisibleGroupMembers(void);
      tEntityDistList               &VisibleEnemies(void);

      void                          AddBehavior(cBehavior &beh);
      void                          RemoveBehavior(cBehavior &beh);

      void                          Set3DRepresentation(RenderLib::CObject*object);
      void                          SetCurrentGroup(cGroup *group);

      D3DXVECTOR3                   Position(void);
      void                          SetPosition(const D3DXVECTOR3 &pos);
      D3DXVECTOR3                   Velocity(void);
      void                          SetVelocity(const D3DXVECTOR3 &vel);
      D3DXQUATERNION                Orientation(void);
      void                          SetVelocity(const D3DXQUATERNION &orient);
      D3DXVECTOR3                   DesiredMove(void);
      void                          SetDesiredMove(const D3DXVECTOR3 &move);

      float                         MaxSpeed(void);
      float                         DesiredSpeed(void);

protected:
      void                          UpdateGroupVisibility(void);
      void                          UpdateEnemyVisibility(void);
      bool                          VisibilityTest(cEntity &otherEntity,
                                                   float &dist);

      cGroup*                       mCurrentGroup;
      tBehaviorList                 mBehaviors;
      cWorld                        &mWorld;
      unsigned                      mFriendMask;
      unsigned                      mEnemyMask;
      unsigned                      mEntityType;

      RenderLib::CObject            *mObject;

      D3DXVECTOR3                   mPosition;
      D3DXVECTOR3                   mVelocity;
```

```
	D3DXQUATERNION		mOrientation;
	D3DXVECTOR3		mDesiredMoveVector;

	float			mSenseRange;
	float			mMaxVelocityChange;
	float			mMaxSpeed;
	float			mDesiredSpeed;
	float			mMoveXScalar;
	float			mMoveYScalar;
	float			mMoveZScalar;

	tEntityDistList		mVisibleGroupMembers;
	tEntityDistList		mVisibleEnemies;
};
```

Here we have the cEntity class. All of the inline implementations were stripped because they are trivial and take up unnecessary space. Refer to the code for the full version. If the cBehaviors are the work horses of the flocking system, the cEntity is the coach driver. This is ultimately the 'thing' that the behaviors will actually be moving around. In our demo, each cEntity is one fish.

```
	cEntity
	(
		cWorld &world,
		unsigned type,
		float senseRange,
		float maxVelocityChange,
		float maxSpeed,
		float desiredSpeed,
		float moveXScalar,
		float moveYScalar,
		float moveZScalar
	);
```

The cEntity constructor takes quite a few parameters. First, it takes a reference to the world so that it can gain access to the list of all of the other groups in the world. Next, it takes a bitmask type to identify what it is. In our demo, there is a player type and a non-player type. This is so that the non-player fish know to run away from the player fish. Next there is a sense range. This value represents how far the entity can see other entities. Any entity farther away than this will be completely ignored. Next there is a max velocity change. This value clamps how quickly the entity can change speeds. This helps prevent instant directional changes and jumps, and keeps the entity moving smoothly. Next is max speed and desired speed. The max speed is the absolute maximum speed the entity can travel, while the desired speed is the speed the entity prefers to travel. The last three scalar values adjust the amount of movement in each of the cardinal directions. This is useful for clamping specific types of movement; for instance, vertical change. Fish tend to swim left and right more than up and down, so the y scalar gets decreased to clamp the amount of vertical movement.

```
	virtual	~cEntity(void);
```

The destructor for cEntity is virtual in order that derived types can be correctly destructed. This is mentioned for every destructor to emphasize its importance. If you do not understand the reason for this, it is highly recommended that you take the Game Institute course C++ Programming for Game

Developers Module I. The cEntity does not own the behaviors it uses, so it does not free them. It also does not own the render object it uses since the scene graph owns that. Thus, the base class cEntity does not free anything, because it does not own anything.

```
	virtual void			Iterate(float timeDelta);
```

The Iterate method applies each of the behaviors which the cEntity has and is the core of the implementation. There are a few other notable methods which the cEntity class exposes, but they are actually called during the course of iteration so we will talk about them as we come across them. Let us take a look at the actual implementation of this method.

```
void cEntity::Iterate(float timeDelta)
{
	mPosition += mVelocity * timeDelta;

	mVisibleGroupMembers.clear();
	mVisibleEnemies.clear();
	UpdateGroupVisibility();
	UpdateEnemyVisibility();

	tBehaviorList::iterator it;
	for (it = mBehaviors.begin(); it != mBehaviors.end(); it++)
	{
		cBehavior *beh = *it;
		beh->Iterate(timeDelta, *this);
	}

	float velChange = D3DXVec3Length(&mDesiredMoveVector);
	if (velChange > mMaxVelocityChange)
	{
		D3DXVec3Normalize(&mDesiredMoveVector, &mDesiredMoveVector);
		mDesiredMoveVector *= mMaxVelocityChange;
	}

	mVelocity += mDesiredMoveVector;

	mVelocity.x *= mMoveXScalar;
	mVelocity.y *= mMoveYScalar;
	mVelocity.z *= mMoveZScalar;

	float speed = D3DXVec3Length(&mVelocity);
	if (speed > mMaxSpeed)
	{
		D3DXVec3Normalize(&mVelocity, &mVelocity);
		mVelocity *= mMaxSpeed;
	}

	D3DXVECTOR3 vec;
	D3DXVec3Normalize(&vec, &mVelocity);
	float pitch = atan2f(-vec.y, sqrtf(vec.z*vec.z + vec.x*vec.x));
	float yaw = atan2f(vec.x, vec.z);
	D3DXQuaternionRotationYawPitchRoll(&mOrientation, yaw, pitch, 0.0f);
}
```

Again, comments were stripped from this listing to compact it as much as possible. As we examine each line of code, a general picture will appear. Refer to the code for the unedited version.

```
	mPosition += mVelocity * timeDelta;
```

The first action of this method is to update our position using the current velocity. This would have to be either the first or absolute last action. In our case, it will be first. What we are doing here is numerically integrating velocity to get our position using standard Euler integration. We use the time delta passed in to avoid frame dependence.

```
	mVisibleGroupMembers.clear();
	mVisibleEnemies.clear();
```

Next, we clear our visibility lists. The cEntity class keeps track of which other entities it can see in its group as well as any enemies it can see. We clear these lists every frame and build them anew.

```
	UpdateGroupVisibility();
```

The next thing we do is update our group visibility list. Let us look at how that is done.

```
void cEntity::UpdateGroupVisibility()
{
	if (mCurrentGroup)
	{
		tEntityList &entities = mCurrentGroup->Entities();
		tEntityList::iterator it;
		for (it = entities.begin(); it != entities.end(); ++it)
		{
			cEntity *e = *it;
			// skip ourselves
			if (e == this)
				continue;

			float dist;
			if (VisibilityTest(*e, dist))
			{
				// keep this list sorted
				pair<float, cEntity*> thepair(dist, e);
				tEntityDistList::iterator pos =
					upper_bound(mVisibleGroupMembers.begin(),
					mVisibleGroupMembers.end(), thepair);
				mVisibleGroupMembers.insert(pos, thepair);
			}
		}
	}
}
```

Here we have the implementation of UpdateGroupVisibility. This method iterates across each of the members of this entity's group, and determines if they can be seen by this entity.

```
	if (mCurrentGroup)
```

First we confirm that we have a group. While all entities should be in a group, checking pointers is always a good idea.

```
            tEntityList &entities = mCurrentGroup->Entities();
            tEntityList::iterator it;
            for (it = entities.begin(); it != entities.end(); ++it)
```

Next, we obtain the list of entities in our group, and begin iterating through them.

```
                  cEntity *e = *it;
                  // skip ourselves
                  if (e == this)
                        continue;
```

If the entity we are currently iterating across is this entity, we skip it, since we do not consider it for visibility.

```
                  float dist;
                  if (VisibilityTest(*e, dist))
```

If the current entity is not this entity, we perform a visibility test.

```
bool cEntity::VisibilityTest(cEntity &otherEntity, float &dist)
{
      // simple test for now, are they close enough that we can "sense them"
      D3DXVECTOR3 distVec = otherEntity.Position() - Position();
      dist = D3DXVec3Length(&distVec);
      if (dist < mSenseRange)
            return(true);
      return(false);
}
```

The visibility test is very straightforward. We simply determine if the entity in question's position is within our sense range. If it is, we can see it, otherwise, we cannot. This method could easily be modified to provide a cone of vision rather than a simple distance check.

```
                        // keep this list sorted
                        pair<float, cEntity*> thepair(dist, e);
                        tEntityDistList::iterator pos =
                              upper_bound(mVisibleGroupMembers.begin(),
                              mVisibleGroupMembers.end(), thepair);
                        mVisibleGroupMembers.insert(pos, thepair);
```

After we determine an entity to be visible, we keep track of its actual distance to us in a pair, and add it to the visible members vector. Notice that the vector is kept sorted. This allows us to easily acquire the closest or farthest entity in the group. We can also perform O $\log_2$ n searches on the list to find a specific entity if we desire.

```
      UpdateEnemyVisibility();
```

After we have updated our group's visibility list, we update our enemy visibility list. Let us take a look at how that is done, since it has a notable difference.

```
void cEntity::UpdateEnemyVisibility()
{
	tGroupList &groups = mWorld.Groups();
	tGroupList::iterator git;
	for (git = groups.begin(); git != groups.end(); ++git)
	{
		cGroup *group = *git;
		tEntityList &entities = group->Entities();
		tEntityList::iterator it;
		for (it = entities.begin(); it != entities.end(); ++it)
		{
			cEntity *e = *it;

			// skip friendly groups
			if ((e->EntityType() & EnemyMask()) == 0)
				break;

			float dist;
			if (VisibilityTest(*e, dist))
			{
				// keep this list sorted
				pair<float, cEntity*> thepair(dist, e);
				tEntityDistList::iterator pos =
					upper_bound(mVisibleEnemies.begin(),
					mVisibleEnemies.end(), thepair);
				mVisibleEnemies.insert(pos, thepair);
			}
		}
	}
}
```

The UpdateEnemyVisibility method obtains the list of groups from the world, and iterates over all of the entities in all of the groups in search of enemy entities.

```
	tGroupList &groups = mWorld.Groups();
	tGroupList::iterator git;
	for (git = groups.begin(); git != groups.end(); ++git)
```

First, the list of groups is obtained from the world, and iteration begins.

```
		cGroup *group = *git;
		tEntityList &entities = group->Entities();
		tEntityList::iterator it;
		for (it = entities.begin(); it != entities.end(); ++it)
```

For each group, the list of entities within the group is obtained, and that list is iterated over.

```
			// skip friendly groups
			if ((e->EntityType() & EnemyMask()) == 0)
				break;
```

Here is the important part. On the assumption that all of the entities within the same group are of the same type (which is safe for this demo, but is not necessarily always the case), if the first entity in the group is not an enemy, the entire group is not an enemy and can be ignored. This is purely an optimization, but it is important to note that the masks of the entities in question are compared with this entity's mask to determine whether it should be considered an enemy.

```
                  float dist;
                  if (VisibilityTest(*e, dist))
                  {
                        // keep this list sorted
                        pair<float, cEntity*> thepair(dist, e);
                        tEntityDistList::iterator pos =
                              upper_bound(mVisibleEnemies.begin(),
                              mVisibleEnemies.end(), thepair);
                        mVisibleEnemies.insert(pos, thepair);
                  }
```

For all those entities determined to be enemies, we do the same visibility test we did for the group mates, and keep track of the distances to the enemy in a sorted list (just as we did with the visible group members list).

```
      tBehaviorList::iterator it;
      for (it = mBehaviors.begin(); it != mBehaviors.end(); it++)
      {
            cBehavior *beh = *it;
            beh->Iterate(timeDelta, *this);
      }
```

Once we have updated our visible friends and enemies lists, we iterate across our list of behaviors and iterate each one. We will go over the implementations of the individual behaviors shortly.

```
      float velChange = D3DXVec3Length(&mDesiredMoveVector);
      if (velChange > mMaxVelocityChange)
      {
            D3DXVec3Normalize(&mDesiredMoveVector, &mDesiredMoveVector);
            mDesiredMoveVector *= mMaxVelocityChange;
      }
```

After we have applied all of our behaviors, we begin some post processing on our desired move vector to bring it in line. First, we determine the length of the desired move vector and ensure it is not bigger than our max velocity change. If it is, we normalize our desired move vector, and set its length (by scaling it) to be the maximum velocity change. In effect, we clamp the vector's length to be that of our max velocity change.

```
      mVelocity += mDesiredMoveVector;
```

Next, we add our desired move vector to our current velocity vector. This will nudge our current velocity vector in the direction of the new desired move.

```
      mVelocity.x *= mMoveXScalar;
```

```
	mVelocity.y *= mMoveYScalar;
	mVelocity.z *= mMoveZScalar;
```

We then apply our Cartesian move scalars. In the case of our demo, we reduce the amount of vertical movement (y axis) so as to produce more lifelike fish movement.

```
	float speed = D3DXVec3Length(&mVelocity);
	if (speed > mMaxSpeed)
	{
		D3DXVec3Normalize(&mVelocity, &mVelocity);
		mVelocity *= mMaxSpeed;
	}
```

Next, we effectively clamp our velocity to be within the bounds of our maximum speed. We do this by first getting the length of our velocity vector, and if that length is greater than the max speed, we normalize the velocity vector and scale the result by our max speed.

```
	D3DXVECTOR3 vec;
	D3DXVec3Normalize(&vec, &mVelocity);
	float pitch = atan2f(-vec.y, sqrtf(vec.z*vec.z + vec.x*vec.x));
	float yaw = atan2f(vec.x, vec.z);
	D3DXQuaternionRotationYawPitchRoll(&mOrientation, yaw, pitch, 0.0f);
```

Lastly, we perform some trigonometric calculations to convert our velocity vector into a quaternion to hold our orientation. Afterwards, we have a new velocity and orientation with which to apply for the next iteration to get a new position.

Before we discuss the movement behaviors, there is one last class definition to investigate.

```
class cBehavior
{
public:
                         cBehavior(void) : mGain(1.0f) {}
	virtual          ~cBehavior(void) {}

	virtual void     Iterate(float timeDelta, cEntity &entity) = 0;

	float            Gain(void) { return(mGain); }
	void             SetGain(float gain) { mGain = gain; }

	virtual string   Name(void) { return("Base Behavior"); }

private:
	float            mGain;

};
```

Here we have the base class cBehavior. This is the interface through which all of the behaviors do their work. Let us go over it in detail.

```
	cBehavior(void) : mGain(1.0f) {}
```

```
      virtual           ~cBehavior(void) {}
```

The default constructor initializes the gain of the behavior to 1.0. The gain value is used to determine the weight upon which the result of the behavior is applied to the entity in question's desired move. The destructor is virtual to allow for correct polymorphic destruction of derived classes.

```
      virtual void      Iterate(float timeDelta, cEntity &entity) = 0;
```

The pure virtual iterate method takes a time delta for the amount of time that has passed since the last call to this method, and the entity to which to apply the behavior's movement decisions.

```
      float             Gain(void) { return(mGain); }
      void              SetGain(float gain) { mGain = gain; }
```

The class provides accessors to allow the gain to be modified post construction.

```
      virtual string    Name(void) { return("Base Behavior"); }
```

The Name method is used by the user interface to properly name the tab of the tab control.

```
      float             mGain;
```

The mGain variable is used to modify the degree to which the behavior will modify the desired behavior of the entity in question. The default is 1.0 which is full effect, while 0.5 would be half effect, and 2.0 would be double effect. The user interface only allows fractional gains.

Now that we have a good understanding of the framework of the application, let us take a look at the implementation of the behaviors implemented in the demo.

3.2.4 Separation

```
class cSeparationBehavior : public cGroupBehavior
{
public:
                        cSeparationBehavior(float sepDist,
                                            float minPercent,
                                            float maxPercent);
      virtual           ~cSeparationBehavior(void);

      virtual void      Iterate(float timeDelta, cEntity &entity);

protected:
      float             mSeparationDistance;
      float             mMinSeparationPercentage;
      float             mMaxSeparationPercentage;
};
```

Here we have the declaration for the Separation Behavior as it exists in our demo. For the sake of brevity, the accessors have been removed from this listing. See the code for the unabridged version.

```
 cSeparationBehavior(float sepDist, float minPercent, float maxPercent);
```

The separation behavior takes three parameters upon construction. The first is the separation distance that the behavior will try to maintain between entities in the group. This applies to entities too close as well as too far away. The goal of the behavior is to ensure all of the entities in the group always stay exactly this distance away from each other. The min percent and max percent values are the minimum and maximum separation percentages that will be applied to limit large changes. The algorithm will determine the actual separation percentage, and then clamp it to these values.

```
      virtual void      Iterate(float timeDelta, cEntity &entity);
```

As with the base class, the Iterate method takes the time passed since the last iteration, and the entity to which to apply the movement. Let us take a closer look.

```
void cSeparationBehavior::Iterate(float timeDelta, cEntity &entity)
{
      tEntityDistList &groupMembers = entity.VisibleGroupMembers();
      if (groupMembers.empty()) return;
      cEntity &nearestGroupMember = *groupMembers.front().second;
      float distanceToClosestGroupMember = groupMembers.front().first;

      float separationPercentage = distanceToClosestGroupMember /
                                   mSeparationDistance;

      D3DXVECTOR3 desiredMoveAdj = nearestGroupMember.Position() -
                                   entity.Position();

      if (separationPercentage < mMinSeparationPercentage)
            separationPercentage = mMinSeparationPercentage;
      if (separationPercentage > mMaxSeparationPercentage)
            separationPercentage = mMaxSeparationPercentage;

      D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
      if (distanceToClosestGroupMember < mSeparationDistance)
      {
            D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
            desiredMoveAdj *= -separationPercentage;
            currentDesiredMove += desiredMoveAdj * Gain();
      }
      else if (distanceToClosestGroupMember > mSeparationDistance)
      {
            D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
            desiredMoveAdj *= separationPercentage;
            currentDesiredMove += desiredMoveAdj * Gain();
      }
      entity.SetDesiredMove(currentDesiredMove);
}
```

The comments have been stripped out for brevity. See the code for the unabridged version. Let us go over this in detail.

```
	tEntityDistList &groupMembers = entity.VisibleGroupMembers();
	if (groupMembers.empty()) return;
```

First, we grab the list of group members. Since this is a group based behavior, if the group is empty, we cannot do anything. Thus, we return.

```
	cEntity &nearestGroupMember = *groupMembers.front().second;
	float distanceToClosestGroupMember = groupMembers.front().first;
```

Next, we get the nearest group member which, on account of sorting, should be at the front of the group members list. We then get the pre-computed distance to that group member.

```
	float separationPercentage = distanceToClosestGroupMember /
	                             mSeparationDistance;
```

Next we compute the separation percentage as the ratio between the actual distance to the closest group member, and the desired separation distance.

```
	D3DXVECTOR3 desiredMoveAdj = nearestGroupMember.Position() -
	                             entity.Position();
```

We also compute a vector from this entity's position to the closest group member.

```
	if (separationPercentage < mMinSeparationPercentage)
	      separationPercentage = mMinSeparationPercentage;
	if (separationPercentage > mMaxSeparationPercentage)
	      separationPercentage = mMaxSeparationPercentage;
```

We clamp the computed separation percentage to our minimum and maximum separation percentages.

```
	D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
```

We grab the entity's current desired move.

```
	if (distanceToClosestGroupMember < mSeparationDistance)
	{
	      D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
	      desiredMoveAdj *= -separationPercentage;
	      currentDesiredMove += desiredMoveAdj * Gain();
	}
```

If the distance to the closest member is less than our desired separation distance, we are too close. Thus, we normalize the vector from this member to our closest member, and scale it by our negated separation percentage. Why negated? The vector we computed was from this entity to the closest member, and we

want a vector going the other way, so we negate it. Lastly, we scale our desired move adjustment by our gain, and add the result to the current desired move.

```
      else if (distanceToClosestGroupMember > mSeparationDistance)
      {
            D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
            desiredMoveAdj *= separationPercentage;
            currentDesiredMove += desiredMoveAdj * Gain();
      }
```

If the distance to the closest member was greater than the separation distance, then we are too far away. Thus, we normalize the vector from this entity to the closest member, and scale it by our separation percentage. We then scale the desired move adjustment vector by our gain, and add the result to the current desired move for the entity. Why do we scale our desired movement vector by our separation percentage? The idea is based on the law of diminishing returns. If you are a great distance from the desired separation distance, you move faster to get there. If you are very close, you move very slowly. This helps settle us into the range we want rather than overshooting all the time.

```
      entity.SetDesiredMove(currentDesiredMove);
```

Finally, we set our entity's desired move to be the newly computed desired move. If we were accurate about the separation distance, then both of the if statements would fail. We would set the desired move to the local value we had cached so nothing would change.

3.2.5 Avoidance

```
class cAvoidanceBehavior : public cBehavior
{
public:
                        cAvoidanceBehavior(float avoidDist, float speed);
      virtual           ~cAvoidanceBehavior(void);

      virtual void      Iterate(float timeDelta, cEntity &entity);

protected:
      float             mAvoidanceDistance;
      float             mAvoidanceSpeed;
};
```

The comments have been removed from this listing for brevity. See the code for the full listing. Let us go over this behavior in detail.

```
      cAvoidanceBehavior(float avoidDist, float speed);
```

The avoidance behavior needs only two parameters; the avoid distance, and a speed. The avoid distance is the distance at which the threat to be avoided will actively be avoided, while the speed is the rate at which the entity will flee from the threat.

```
      virtual void      Iterate(float timeDelta, cEntity &entity);
```

As with all of the behaviors, the Iterate method takes the time since the last iteration, and a reference to the entity to which the movement is to be applied. Let us go over this implementation.

```
void cAvoidanceBehavior::Iterate(float timeDelta, cEntity &entity)
{
      tEntityDistList &enemies = entity.VisibleEnemies();
      if (enemies.empty()) return;
      cEntity &nearestEnemy = *enemies.front().second;
      float nearestEnemyDist = enemies.front().first;

      // head away from the enemy
      if (nearestEnemyDist < mAvoidanceDistance)
      {
            D3DXVECTOR3 desiredMoveAdj = entity.Position() -
                                         nearestEnemy.Position();
            D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
            D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
            desiredMoveAdj *= mAvoidanceSpeed;  // move away
            currentDesiredMove += desiredMoveAdj * Gain();
            entity.SetDesiredMove(currentDesiredMove);
      }
}
```

The implementation is fairly straightforward -- we find the closest visible enemy, and run the other way.

```
      tEntityDistList &enemies = entity.VisibleEnemies();
      if (enemies.empty()) return;
```

First and foremost, we get the list of visible enemies. If there are no visible enemies, we do nothing.

```
      cEntity &nearestEnemy = *enemies.front().second;
      float nearestEnemyDist = enemies.front().first;
```

If there are visible enemies, then we get the first one in the list which has been sorted, and we fetch the distance to that enemy.

```
      if (nearestEnemyDist < mAvoidanceDistance)
```

If that distance is less than our avoidance distance, we run away.

```
            D3DXVECTOR3 desiredMoveAdj = entity.Position() -
                                         nearestEnemy.Position();
```

First we compute a vector from the enemy to us.

```
            D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
```

Then we grab our current desired move vector.

```
            D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
            desiredMoveAdj *= mAvoidanceSpeed;  // move away
            currentDesiredMove += desiredMoveAdj * Gain();
```

We then normalize our enemy-to-us vector, scale it by our movement speed, apply our gain, and add the result to the current desired move.

```
            entity.SetDesiredMove(currentDesiredMove);
```

Lastly, we set the current desired move to our newly computed desired move.

3.2.6 Cohesion

```
class cCohesionBehavior : public cGroupBehavior
{
public:
                              cCohesionBehavior(float turnRate);
      virtual                 ~cCohesionBehavior(void);

      virtual void      Iterate(float timeDelta, cEntity &entity);

protected:
      float             mTurnRate;
};
```

As usual, the comments have been removed from this listing for brevity. See the code for the full listing. Let us go over this in detail.

```
      cCohesionBehavior(float turnRate);
```

The cohesion behavior takes a single parameter -- the turn rate. The turn rate is the maximum rate at which the entity will change direction in order to head towards the average center of the group.

```
      virtual void      Iterate(float timeDelta, cEntity &entity);
```

The Iterate method takes the time passed since the last iteration, and a reference to the entity for which to generate a movement. Let us look at this implementation.

```
void cCohesionBehavior::Iterate(float timeDelta, cEntity &entity)
{
      tEntityDistList &groupMembers = entity.VisibleGroupMembers();
      if (groupMembers.empty()) return;

      // compute center of mass of the group
      D3DXVECTOR3 groupCenterOfMass(0.0f, 0.0f, 0.0f);
      tEntityDistList::iterator it;
      for (it = groupMembers.begin(); it != groupMembers.end(); ++it)
      {
```

```
            cEntity *e = (*it).second;
            groupCenterOfMass += e->Position();
      }
      groupCenterOfMass /= (float)groupMembers.size();

      // move towards the center of the group
      D3DXVECTOR3 desiredMoveAdj = groupCenterOfMass - entity.Position();
      D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
      D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
      desiredMoveAdj *= mTurnRate;
      currentDesiredMove += desiredMoveAdj * Gain();
      entity.SetDesiredMove(currentDesiredMove);
}
```

The cohesion behavior iterates across the list of visible group members, and computes the group's average center of mass. It then generates a vector towards this center.

```
      tEntityDistList &groupMembers = entity.VisibleGroupMembers();
      if (groupMembers.empty()) return;
```

First, the list of visible group members is obtained. If there are no visible group members, we return, as there is no group center of mass.

```
      D3DXVECTOR3 groupCenterOfMass(0.0f, 0.0f, 0.0f);
      tEntityDistList::iterator it;
      for (it = groupMembers.begin(); it != groupMembers.end(); ++it)
      {
            cEntity *e = (*it).second;
            groupCenterOfMass += e->Position();
      }
      groupCenterOfMass /= (float)groupMembers.size();
```

Next, we iterate across the visible group members list, and sum up the positions of each member. We then divide out the number of group members that were visible to obtain the average group position, or center of perceived mass.

```
      D3DXVECTOR3 desiredMoveAdj = groupCenterOfMass - entity.Position();
```

We then compute a vector towards that center.

```
      D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
```

And obtain our current desired move.

```
      D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
      desiredMoveAdj *= mTurnRate;
      currentDesiredMove += desiredMoveAdj * Gain();
```

We normalize the us-to-center vector, and scale it by our maximum turn rate. Next we apply our gain and add the result to our current desired move.

```
	entity.SetDesiredMove(currentDesiredMove);
```

Finally, we set our entity's desired move to our newly computed desired move.

3.2.7 Alignment

```
class cAlignmentBehavior : public cGroupBehavior
{
public:
                          cAlignmentBehavior(float turnRate);
	virtual           ~cAlignmentBehavior(void);

	virtual void      Iterate(float timeDelta, cEntity &entity);

protected:
	float             mTurnRate;
};
```

The comments have been removed from this listing for brevity. Look to the code for the full listing. Let us go over this behavior in detail.

```
	cAlignmentBehavior(float turnRate);
```

Like the cohesion behavior, the alignment behavior takes only a single parameter, the turn rate. The turn rate parameter limits the rate at which the entity will change direction in an effort to match heading with its visible group mates.

```
	virtual void      Iterate(float timeDelta, cEntity &entity);
```

As in all the other behaviors, the Iterate method takes the time passed since the last iteration and the entity upon which to add desired move. Let us examine this implementation.

```
void cAlignmentBehavior::Iterate(float timeDelta, cEntity &entity)
{
	tEntityDistList &groupMembers = entity.VisibleGroupMembers();
	if (groupMembers.empty()) return;
	cEntity &nearestGroupMember = *groupMembers.front().second;

	// match the heading of our closest group member
	D3DXVECTOR3 desiredMoveAdj = nearestGroupMember.Velocity();
	D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
	D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
	desiredMoveAdj *= mTurnRate;
	currentDesiredMove += desiredMoveAdj * Gain();
	entity.SetDesiredMove(currentDesiredMove);
}
```

In this implementation, we elected to use the heading of the nearest group member to align an entity, rather than an average of all of the visible group members. It would be simple enough to make the modification to average all of the headings of all of the visible group members and use that rather than only the closest member. You can try this as an exercise if you wish.

```
tEntityDistList &groupMembers = entity.VisibleGroupMembers();
if (groupMembers.empty()) return;
```

First, we get the list of visible group members. If that list is empty, we return. We cannot align without other members to align with.

```
cEntity &nearestGroupMember = *groupMembers.front().second;
```

Next we obtain the closest group member by virtue of our sorted list.

```
D3DXVECTOR3 desiredMoveAdj = nearestGroupMember.Velocity();
```

We then obtain the velocity of our nearest group member. The velocity embodies the direction which the member is facing.

```
D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
```

We then obtain our current desired move.

```
D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
desiredMoveAdj *= mTurnRate;
currentDesiredMove += desiredMoveAdj * Gain();
```

Next we normalize the velocity obtained from our closest group member, and scale it by our turn rate. This is now our desired move adjustment. We apply our gain, and add the result to the current desired move.

```
entity.SetDesiredMove(currentDesiredMove);
```

Finally, we set this entity's desired move to be our newly computed desired move.

3.2.8 Cruising

We have now covered all of the normal behaviors used in flocking applications, but there are more behaviors yet to investigate in our demo. The first is the Cruising behavior which, as mentioned earlier, is the behavior which decides where the entity would go if the decision were based solely on that entity. This is useful when it cannot see any of its group mates and needs to decide where to go.

```
class cCruisingBehavior : public cBehavior
{
public:
                              cCruisingBehavior
                              (
                                    float randMoveXChance,
                                    float randMoveYChance,
                                    float randMoveZChance,
                                    float minRandomMove,
                                    float maxRateChange,
                                    float minRateChange
                              );
      virtual                 ~cCruisingBehavior(void);

      virtual void            Iterate(float timeDelta, cEntity &entity);

protected:
      float                   mRandMoveXChance;
      float                   mRandMoveYChance;
      float                   mRandMoveZChance;
      float                   mMinRandomMove;
      float                   mMaxRateChange;
      float                   mMinRateChange;
};
```

The comments have been removed from this listing for brevity. Look to the code for the full listing. Let us go over this behavior in detail.

```
      cCruisingBehavior
      (
            float randMoveXChance,
            float randMoveYChance,
            float randMoveZChance,
            float minRandomMove,
            float maxRateChange,
            float minRateChange
      );
```

The cruising behavior takes quite a few parameters. The first three are percent chances the entity will decide to move in one of the cardinal directions. Next, we have the min random move, which is the minimum amount the entity will decide to move in the direction chosen. Last, we have the max and min rate changes, which limit the amount the entity is allowed to change direction.

```
      virtual void      Iterate(float timeDelta, cEntity &entity);
```

The Iterate method takes the time passed since the last iteration, and a reference to the entity upon which the cruising shall occur. Let us have a look at the implementation.

```
void cCruisingBehavior::Iterate(float timeDelta, cEntity &entity)
{
      // determine how fast we are going vs how fast
      // we would like to be going
```

```
      float currentSpeed = D3DXVec3Length(&entity.Velocity());
      float percentDesiredSpeed = fabs((currentSpeed - entity.DesiredSpeed()) /
                                       entity.MaxSpeed());
      float signum = (currentSpeed - entity.DesiredSpeed()) > 0? -1.0f : 1.0f;

      // clamp rate changes
      if (percentDesiredSpeed < mMinRateChange)
            percentDesiredSpeed = mMinRateChange;
      if (percentDesiredSpeed > mMaxRateChange)
            percentDesiredSpeed = mMaxRateChange;

      // add some random movement
      D3DXVECTOR3 desiredMoveAdj(0.0f, 0.0f, 0.0f);
      float randmove = (float)rand() / (float)RAND_MAX;
      if (randmove < mRandMoveXChance)
            desiredMoveAdj.x += mMinRandomMove * signum;
      else if (randmove < mRandMoveYChance)
            desiredMoveAdj.y += mMinRandomMove * signum;
      else if (randmove < mRandMoveZChance)
            desiredMoveAdj.z += mMinRandomMove * signum;

      D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
      D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
      desiredMoveAdj *= mMinRateChange * signum;
      currentDesiredMove += desiredMoveAdj * Gain();
      entity.SetDesiredMove(currentDesiredMove);
}
```

The algorithm performs straightforward random picking of which direction to go next, and applies it.

```
      float currentSpeed = D3DXVec3Length(&entity.Velocity());
```

First, we determine how fast we are going at the moment.

```
      float percentDesiredSpeed = fabs((currentSpeed - entity.DesiredSpeed()) /
                                       entity.MaxSpeed());
```

We then determine what percentage of the entity's desired speed we have achieved, as a ratio of its max speed.

```
      float signum = (currentSpeed - entity.DesiredSpeed()) > 0? -1.0f : 1.0f;
```

Next we establish whether we are going faster or slower than our desired speed, and store off a signum multiplier. By doing this, we can easily flip the direction of any directional vectors we create.

```
      if (percentDesiredSpeed < mMinRateChange)
            percentDesiredSpeed = mMinRateChange;
      if (percentDesiredSpeed > mMaxRateChange)
            percentDesiredSpeed = mMaxRateChange;
```

Next, we clamp the percent desired speed to our rate limits. This will prevent us from stopping instantly or jumping to full tilt from a standstill.

```
	float randmove = (float)rand() / (float)RAND_MAX;
	if (randmove < mRandMoveXChance)
		desiredMoveAdj.x += mMinRandomMove * signum;
	else if (randmove < mRandMoveYChance)
		desiredMoveAdj.y += mMinRandomMove * signum;
	else if (randmove < mRandMoveZChance)
		desiredMoveAdj.z += mMinRandomMove * signum;
```

Now we decide which direction we want to move by generating a random number, and comparing it against our percent chances per cardinal direction. In the case of the demo, the default is to limit the chance to start going up or down in the y dimension to keep the motion more realistic.

```
	D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
```

We grab our current desired move.

```
	D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
	desiredMoveAdj *= mMinRateChange * signum;
	currentDesiredMove += desiredMoveAdj * Gain();
```

We then normalize our randomly generated move vector, and scale it by our minimum rate change to get us going in the direction desired. We multiply by our signum for some extra randomness, apply our gain, and add the result to the current desired move.

```
	entity.SetDesiredMove(currentDesiredMove);
```

Finally, we set our entity's desired move using our newly calculated desired move.

3.2.9 Stay Within Sphere

The last behavior implemented in this demo is the Stay Within Sphere behavior. This behavior was written to prevent the necessity of having to teleport the fish to keep them around the player's location. Thus, they will just turn around when they get too far away. Let us take a look at the implementation.

```
class cStayWithinSphereBehavior : public cBehavior
{
public:
                        cStayWithinSphereBehavior(const D3DXVECTOR3 &center,
                                                  float radius);
	virtual             ~cStayWithinSphereBehavior(void);

	virtual void        Iterate(float timeDelta, cEntity &entity);

protected:
	D3DXVECTOR3         mCenter;
	float                     mRadius;
};
```

The comments have been removed from this listing for brevity. Look to the code for the full listing. Let us go over this in detail.

```
      cStayWithinSphereBehavior(const D3DXVECTOR3 &center,  float radius);
```

The stay within sphere behavior takes a center for the sphere, and a radius for the sphere's radius.

```
      virtual void      Iterate(float timeDelta, cEntity &entity);
```

This behavior uses the Iterate method, which takes the time from the last iteration and the entity to keep within the sphere. Let us see how it does this.

```
void cStayWithinSphereBehavior::Iterate(float timeDelta, cEntity &entity)
{
      D3DXVECTOR3 toCenter = mCenter - entity.Position();
      float dist = D3DXVec3Length(&toCenter);
      if (dist > mRadius)
      {
            D3DXVECTOR3 desiredMoveAdj = toCenter / dist;
            D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
            D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
            desiredMoveAdj *= entity.MaxSpeed();
            currentDesiredMove += desiredMoveAdj * Gain();
            entity.SetDesiredMove(currentDesiredMove);
      }
}
```

The behavior is pretty simple. It determines the distance from the center of the sphere to the entity. If that distance puts the entity outside the sphere, it moves the entity towards the center of the sphere. Let us take a closer look.

```
      D3DXVECTOR3 toCenter = mCenter - entity.Position();
      float dist = D3DXVec3Length(&toCenter);
```

First, the distance from the entity to the center of the sphere is computed.

```
      if (dist > mRadius)
```

If that distance is greater than the radius of the sphere, we are outside the bounds of the sphere.

```
            D3DXVECTOR3 desiredMoveAdj = toCenter / dist;
```

If we are outside the sphere, we first compute a desired move adjustment by taking the vector from the entity to the center of the sphere, and divide out the distance to the center.

```
            D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
```

We then get the current desired movement vector.

```
            D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
            desiredMoveAdj *= entity.MaxSpeed();
            currentDesiredMove += desiredMoveAdj * Gain();
```

Then we normalize the desired movement adjustment, and scale it by the maximum speed of the entity. We want to be within the sphere. We apply our gain and add the result to the current desired move.

```
            entity.SetDesiredMove(currentDesiredMove);
```

Finally, we set our entity's desired move to be the newly computed desired move.

At this stage we now have a fairly complete flocking demonstration. There is plenty to learn here so it would be best if you took the needed to time really examine the source code for the demo in detail.

Conclusion

In this chapter we made the transition from pathfinding to decision making. We have found that flocking can be a useful means to simulate real life behavior of groups which move together in a fashion where they are directly influenced by other nearby entities. This type of behavior is found in real life in flocks of birds, schools of fish, herds of cows and other livestock, as well as crowds of people. You can probably imagine lots of scenarios where the behaviors we studied could be applied in game situations.

Even at a simple level you could imagine a flock consisting of only two members – the player and his buddy. That buddy could use any number of these behaviors to stay within a certain distance and travel in roughly the same direction as the player. If attacked, your buddy might have to break away to defend himself, but if he has to stay within a given sphere around the player, ultimately flee if the player decides to flee. Lots of interesting ideas abound here and the results on screen can be very compelling.

Obviously you could continue to extend even this simple simulation just by adding other buddies to your flock. Pretty soon you find yourself with some crude but convincing squad-like behavior. The squad-like behavior can be made much more effective with the addition of a robust decision making architecture for each entity (and even for the group itself) and some proper environmental awareness and navigation. And of course, you can apply these same ideas to groups of enemies as well (e.g. a group of Orcs in an RPG game).

But even just for getting groups of entities to navigate around in the environment and maintain some semblance of cohesion (e.g., in a real-time strategy game) this system can be very helpful. Your flock can be loose and somewhat randomly distributed over a given area or you can have it maintain tight unit formation. Just create a new behavior (e.g., a grid-based concept for more rigid formations) and you are ready to go. As our fish demo demonstrates, flocking behaviors are useful in any environment. You can use it on land to gather your armies, at sea to manage your naval forces, or even in outer space to manage your armada of various spacecraft.

To summarize, flocking makes use of Behavior Based Movement, which effectively sums up the desired movements from several behavioral algorithms to produce a final desired movement. The methods we discussed were:

- Separation – Keeping the entity at a given distance from all of its neighboring group mates.
- Cohesion – Keeping the entity in line with the center of mass of the overall group, thereby giving the appearance that the entity has a preference to be near the group.
- Alignment – Keeping the entity aligned in its orientation with the rest of the group (or at least its closest group mate) so as to keep it traveling in the same direction as the group as a whole.
- Avoidance – Keeping the entity out of harm from dangerous entities.
- Cruising – Giving the entity a will of its own when it has no nearby group mates to guide it.

Flocking sets the stage for further exploration of decision making in that it is ultimately all about deciding where an entity wants to go. Decision making itself is the core of artificial intelligence, as it is the means for making something look or feel intelligent. It binds the other types of artificial intelligence together, as it makes use of the classification systems to make its decisions, and also makes decisions to move to desired destinations, which of course requires pathfinding.

In the next chapter, we will discuss one of the most flexible decision making systems available to the game AI developer: the finite state machine. Moreover, we will examine scripting and how its use can extend our finite state machines. Finally, we will look at our finite state machine application, which allows us to create our own custom state machines, simulate them, and save them for use in our games.